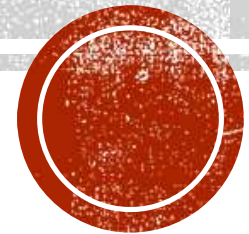


# WHY REST?



# PROBLEMS WITH WSDL-SOAP

- ✦ XML can become very verbose
  - ✦ So there is a tradeoff between bandwidth, data transfer, etc
- ✦ Primarily for heavy duty work
  - ✦ Can transfer binary data through attachments
- ✦ XML is designed for machine readability
  - ✦ Tedious to debug
  - ✦ Even though tools are available
- ✦ You don't need
  - ✦ A supercomputer for browsing facebook
  - ✦ A written contract and registrar for borrowing 10 Rs from a friend
- ✦ The necessity of webservises have become widespread
  - ✦ Unlike earlier we can imagine using webservises even for simpler things
  - ✦ The advent of smartphones have exacerbated this



# REST TO THE RESCUE

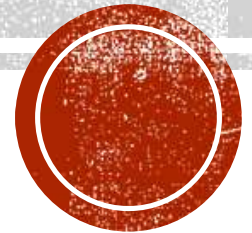


- ✦ Piggybacks on HTTP verbs (we will soon see how)
- ✦ JSON format though not directly connected to REST, it goes hand-in-hand with the proliferation of RESTful services
- ✦ JSON is a pleasure to deal with (Thanks Doug Crockford)
  - ✦ (read) Javascript the good parts
- ✦ Cons
  - ✦ No machine readable definitions (at least till a while back)
    - ✦ Remember I am old!
  - ✦ Hence, No auto *stub* generation tools
    - ✦ But we don't need them anyway
      - ✦ because its simple



# REPRESENTATIONAL STATE TRANSFER (REST)

The theory of Representational State Transfer – The  
Architectural Style – By Roy Fielding



UNIVERSITY OF CALIFORNIA,  
IRVINE

Architectural Styles and the Design of Network-based Software Architectures

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Roy Thomas Fielding

## REST

PhD Dissertation of Roy Fielding (2000) at UCI. He was also the chairman of Apache Software foundation and a prolific contributor for WWW as member of W3C. He is one of the authors of **HTTP/1.1** specification. Currently Principal Scientist in Adobe





# SOFTWARE DESIGN VS NETWORKING RESEARCH

- Software Design research
  - the categorization of software designs and the development of design methodologies
  - but **NOT** the impact of various **design choices on system behavior**
- Networking research
  - Generic communication behavior between systems
  - Improving the performance of particular communication techniques
  - But **NOT** the fact that changing the **interaction style of an application** can have more impact on performance than the communication protocols.



# FIELDING'S FOCUS

“ My work is motivated by the desire to understand and evaluate the **architectural design** of **network-based application software** through principled use of **architectural constraints**, thereby obtaining the ...properties desired of an architecture. ”



# SOFTWARE ARCHITECTURE

“A software architecture is an **abstraction** of the **run-time elements** of a software system during some **phase** of its operation. A system may be composed of many levels of abstraction and many phases of operation, each with its own software architecture.”





# SOFTWARE ARCHITECTURE - ABSTRACTION

- At the heart of software architecture is the principle of abstraction
  - A complex system will contain many levels of abstraction, each with its own architecture.
- Architectural elements are delineated by the abstract interfaces
  - Within each element may be found another architecture
- The *sub*-architecture
  - Defines the system of sub-elements that implement the behavior represented by the parent element's abstract interface



# SOFTWARE ARCHITECTURE - PHASES

- In addition, a software system will often have multiple operational phases
  - start-up
  - Initialization
  - normal processing
  - re-initialization
  - shutdown.
- Each operational phase has its own architecture
- For example, a configuration file will be treated as a data element during the start-up phase, but won't be considered otherwise
- An overall description of a system architecture should also describe the architecture of transitions between phases



# SOFTWARE ARCHITECTURE - ELEMENTS

- Perry and Wolf [105] define processing elements as “transformers of data”
- Shaw et al. [118] describe components as “the locus of computation and state.”

“A component is a unit of software that performs some function at run-time. Examples include programs, objects, processes, and filters.”

“A software architecture is defined by a configuration of architectural elements: **components, connectors, and data**—constrained in their relationships in order to achieve a desired set of architectural properties.”

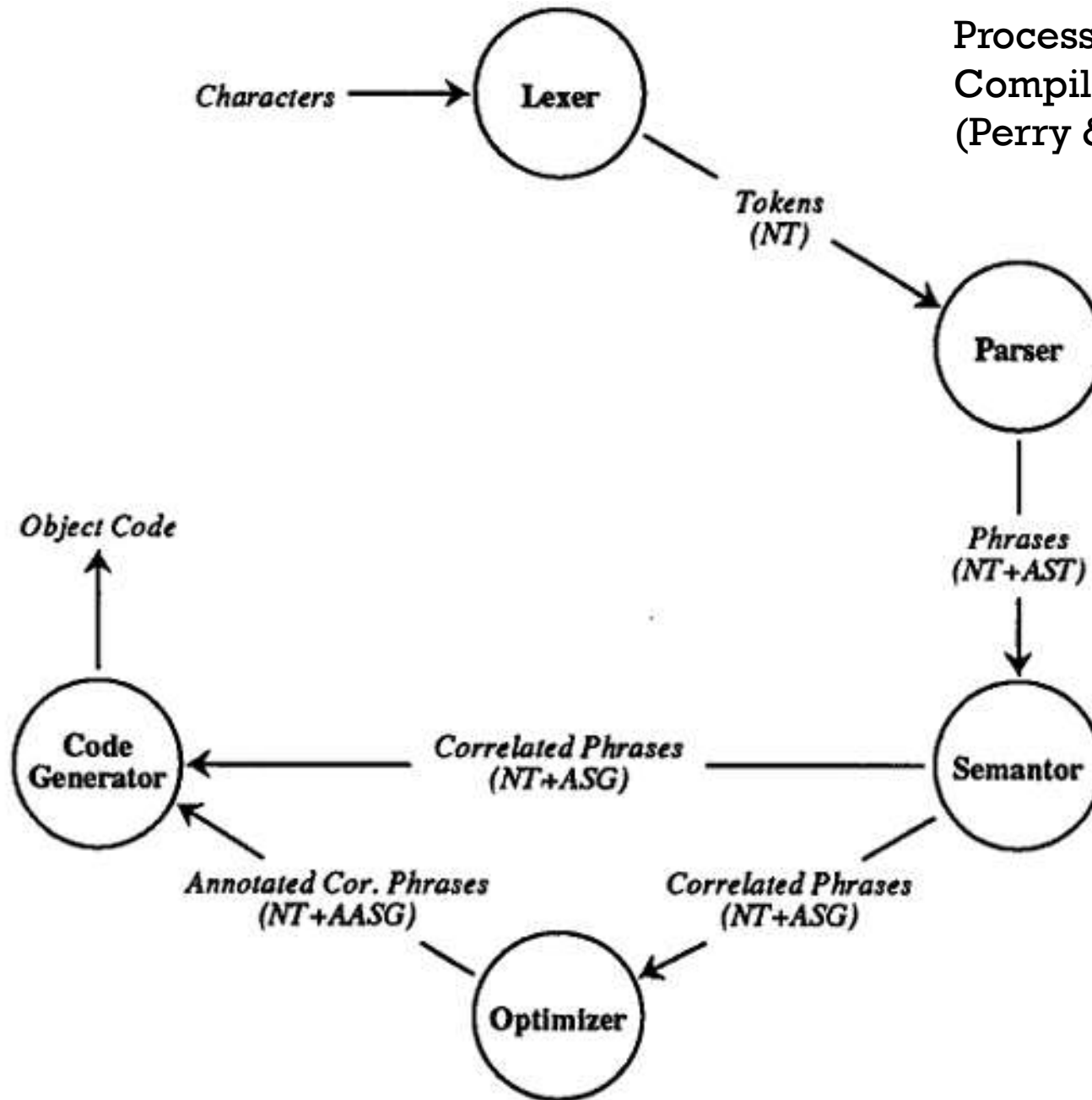


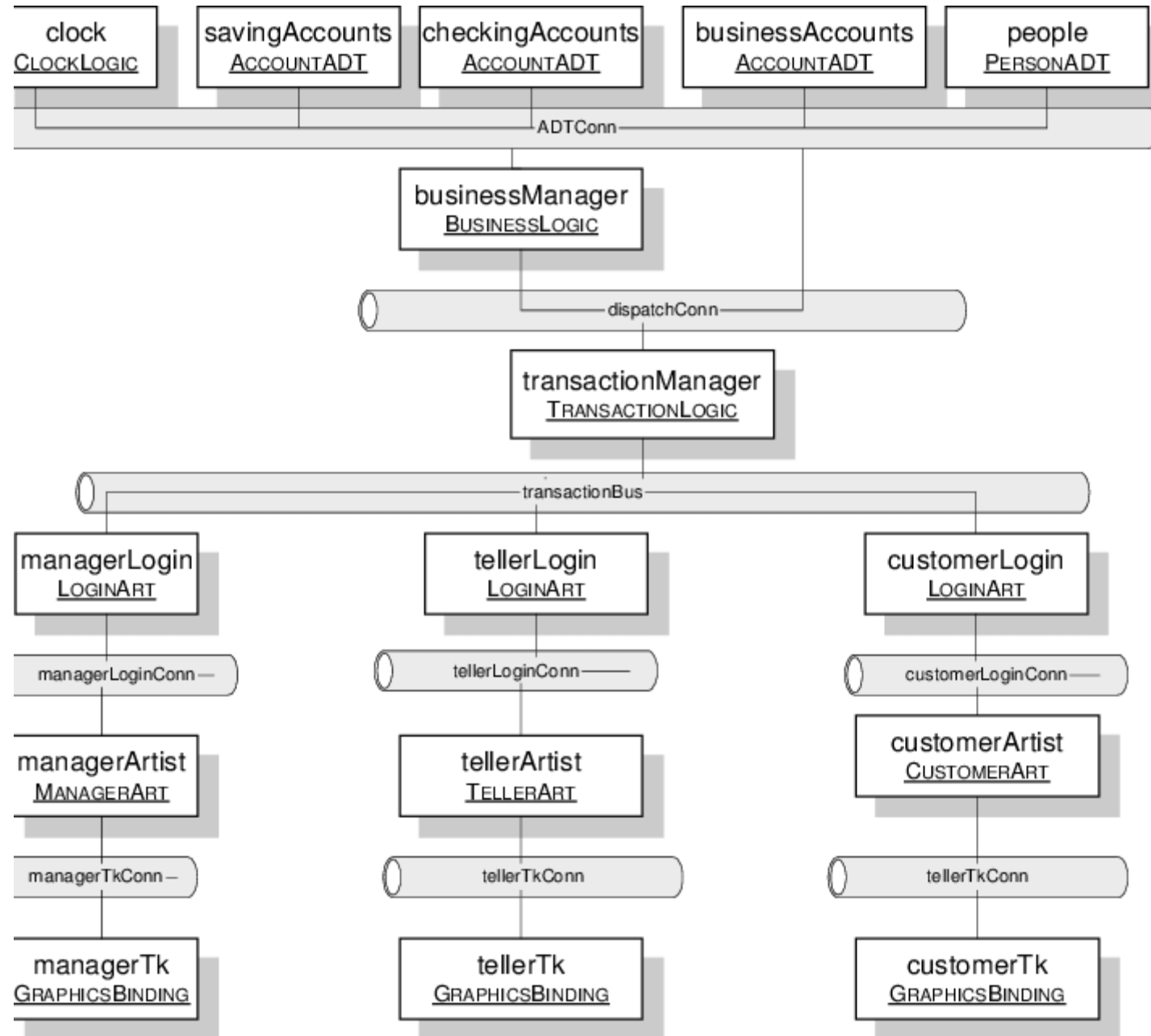
# COMPONENTS, CONNECTORS & DATA

1. A **component** is an abstract unit of software instructions and internal state that provides a transformation of data via its interface.
2. A **connector** is an abstract mechanism that mediates communication, coordination, or cooperation among components.
3. A **datum** is an element of information that is transferred from a component, or received by a component, via a connector.



Processing View of Sequential  
Compiler Architecture  
(Perry & Wolf 1992)







# SOFTWARE ARCHITECTURE VS SOFTWARE STRUCTURE

- There is an important distinction between software architecture and software structure
- **Software Architecture** is an abstraction of the run-time behavior of a software system.
- **Software Structure** on the other hand is a property of the static software source code.
- Sometimes the structure may correspond well with the architecture.
  - But, not always.
  - Also, this is not a necessary condition



# LET'S DISCUSS

- Describe an NLP Software (assumed)



# ARCHITECTURAL PROPERTIES

- The set of architectural properties includes all properties that derive from the selection and arrangement of components, connectors, and data.
- Functional properties (Application behavior specification)
- Non-Functional properties (ease of use, efficiency, etc)
- Properties are achieved via architectural constraints (we will see next slide)
- The goal of **architectural design** is to create an architecture with a set of architectural properties that form a **superset** of the system (functional & non-functional) requirements.



# ARCHITECTURAL CONSTRAINTS

- Constraints are often motivated by Software design principles/patterns.
- Remember these.... constraints
  1. The model is **only responsible** for managing the data of the application. It receives user input from the controller.
  2. The view is **only responsible** for rendering presentation of the model in a particular format.
  3. The controller **is only responsible** for responds to the user input and performs interactions on the data model objects.
- If you follow all these constraints, then it can be said that you are adhering to **Model-View-Controller** Design pattern.



# ARCHITECTURAL STYLE

- There is no authority. Another name may be “Architectural pattern”

“An architectural style is a **coordinated set of architectural constraints** that restricts the roles/features of architectural elements and the allowed relationships among those elements within any architecture that conforms to that style.”



# NETWORK BASED VS DISTRIBUTED

- Distributed system is one that looks to its users like an ordinary centralized system, but runs on multiple, independent CPUs.
- Network-based systems are those capable of operation across a network
  - but not necessarily transparent to the user.
  - It may even be desirable for user to be aware of the difference between an action that requires a network request
- The focus of Roy Fielding is about **Network Based Application Architectures**





# NETWORK BASED ARCHITECTURE- DATA FLOW STYLES

- Pipe and Filter style (PF)
  - Each component (filter) reads streams of data on its inputs and produces streams of data on its outputs
  - Usually while applying a transformation to the input streams and processing them incrementally
  - So output begins before the input is completely consumed
- Uniform Pipe and Filter style (UPF)
  - Adds the constraint that all filters must have the same interface.
  - Example: Unix (ish) operating system styles



# EXAMPLE OF STYLE INDUCED PROPERTIES (PF & UPF)


- PF allows the designer to understand the overall input/output of the system as a simple composition of the behaviors of the individual filters (**simplicity**).
- PF supports reuse: any two filters can be hooked together (**reusability**)
- PF systems can be easily maintained and enhanced: new filters can be added to existing systems (**extensibility**)
- In UPF, Restricting the interface allows independently developed filters to be arranged at will to form new applications

Style	Derivation	Net Perform.	UP Perform.	Efficiency	Scalability	Simplicity	Evolvability	Extensibility	Customiz.	Configur.	Reusability	Visibility	Portability	Reliability
PF			±			+	+	+		+	+			
UPF	PF	-	±			++	+	+		++	++	+		

Table 3-1. Evaluation of Data-flow Styles for Network-based Hypermedia



# NETWORK BASED ARCHITECTURE- REPLICATION STYLES

- Replicated Repository Style (RRS)
  - Improve the accessibility of data and scalability of services by having more than one process provide the same service
  - Provide clients the illusion that there is just one centralized service
- Cache Style (\$) 
  - A variant of replicated
  - Replication of the result (response) of an individual request such that it may be reused by later requests

# NETWORK BASED ARCHITECTURE- HIERARCHICAL STYLES

- Client Server Style (CS)

- A server component, offering a set of services, listens for requests upon those services.
- A client component, desiring that a service be performed, sends a request to the server via a connector.

- Layered Client Server Style (LCS)

- Each layer providing services to the layer above it and using services of the layer below it
- Reduce coupling across multiple layers by hiding the inner layers from all except the adjacent outer layer
- Thus improving evolvability and reusability.
- Layered-client-server adds proxy and gateway components to the client-server style.



# NETWORK BASED ARCHITECTURE- HIERARCHICAL STYLES

- Client Stateless Server Style (CSS)
  - Derives from client-server with the additional constraint that no session state is allowed on the server component.
  - Each request from client to server must contain all of the information necessary to understand the request.
  - Session state is kept entirely on the client.
- Client-Cache-Stateless-Server (C\$SS)
  - Derives from the CSS and adds the \$ style
  - A cache acts as a mediator between client and server
  - The responses (if cacheable) can be reused



# NETWORK BASED ARCHITECTURE- HIERARCHICAL STYLES

- Layered-Client-Cache-Stateless-Server (LC\$CSS)
  - Derives from both LCS and C\$SS through the addition of proxy and/or gateway components.
- As you can see, when styles have **non-conflicting set of constraints** they can be stacked up to create new styles





# NETWORK BASED ARCHITECTURE- MOBILE CODE STYLES

- Virtual Machine (**VM**)
  - The code must be executed in some fashion, preferably within a controlled environment to satisfy security and reliability concerns
- Remote Evaluation (**REV**)
  - Derived from the client-server (CS) and virtual machine (VM) styles
  - A client component has the know-how necessary to perform a service
  - But client lacks the resources (CPU cycles, data source, etc.) required
  - The client sends the know-how to a server component at the remote site
  - The server executes the code using the resources available



# NETWORK BASED ARCHITECTURE- MOBILE CODE STYLES

- Code on Demand (COD)
  - A client component has access to a set of resources, but not the know-how
  - The client sends a request to a remote server for the code representing that know-how
  - The server sends it and client receives that code, and executes it locally



# NETWORK BASED ARCHITECTURE- MOBILE CODE STYLES

- Layered-Code-on-Demand-Client-Cache-Stateless-Server (**LCODC\$SS**)
  - Derived by incorporating code-on-demand to the layered-client-cache-stateless-server (LC\$SS)
  - Example Java Applets (extinct – don't use)



# NETWORK BASED ARCHITECTURE- OTHER STYLES

- Other Hierarchical Styles
  - Remote session (example: cookies)
  - Remote Data Access (example: JDBC)
- Other Mobile Code Styles
  - Mobile Agent
- Peer to Peer Style
  - Event Based Integration
  - C2 Architectural Style
- Blackboard architectural style
- Many more....But we will focus on the ones that are important for ***deriving*** the REST framework



# REPRESENTATIONAL STATE TRANSFER (REST)

- REST is an **architectural style** for **Distributed Hypermedia System**
- REST is a hybrid style **derived** from several of the network-based architectural styles seen so far
- With one additional (Most) important constraint: **Uniform Connector Interface**



# DERIVING REST

- The first constraints added to our hybrid style are those of the client-server architectural style (CS)
  - Separation of concerns is the principle behind the client-server
  - Separation allows the components to evolve independently
  - Separating UI responsibility leads to improved portability (multiplatform)
- Next we improve on this to Client Stateless Server style (CSS)
  - Each request should contain all the server needs
  - Session state should be maintained entirely on the client
  - This constraint induces the properties of visibility, reliability, and scalability.





# DERIVING REST

- Next we improve on this to Client Cache Stateless Server style (C\$SS)
  - If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
  - This improves efficiency, scalability, and user-perceived performance
- Next we improve on this to LC\$SS
  - Layered system style allows an architecture to be composed of hierarchical layers
  - Each component cannot “see” beyond the immediate layer
  - This bounds the overall system complexity and promotes substrate independence.
  - Layers can be used to encapsulate legacy services and to protect new services from legacy clients
- The central feature that distinguishes REST is the Uniform Interface style, which we will discuss separately in the next slide



# DERIVING REST

- Next we add Code on Demand Style to derive (LCODC\$CSS)
  - This allows remote code distribution.
  - Remember REST is a general architectural style applicable for Networked Hypermedia application software.
  - We are only interested in REST style as applied to API/Services design, so we will not focus more on this.
- Finally, the central feature that distinguishes REST is the Uniform Interface style, which we will discuss separately in the next slide



# UNIFORM INTERFACE STYLE

- Emphasizes a uniform interface between components
- The overall system architecture is simplified and the visibility of interactions is improved
- Trade-off: degrades efficiency
  - Information is transferred in a standardized form rather than one which is specific to the application's needs.
- In order to get a Uniform interface, this is further decomposed into four constraints
  - REST interface constraints



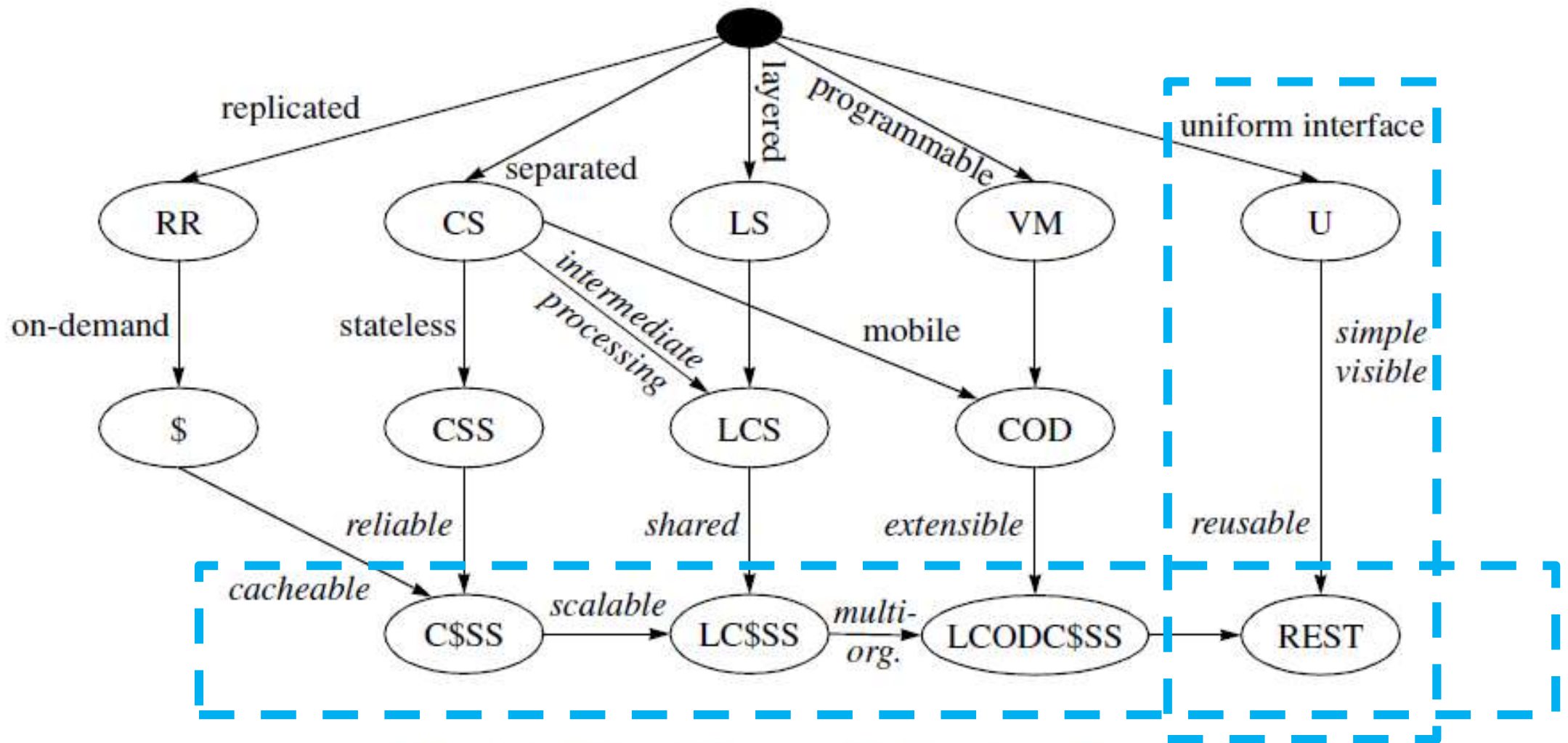


Figure 5-9. REST Derivation by Style Constraints



# REST INTERFACE CONSTRAINTS

1. Identification of resources
2. Manipulation of resources through representations
3. Self-descriptive messages
4. Hypermedia as the engine of application state



# REST RESOURCE IDENTIFICATION

- The key abstraction of information in REST is a **resource**.
- Any information that can be named can be a resource:
  - a document or image,
  - a temporal service (e.g. “today’s weather in Los Angeles”)
- REST uses a resource identifier to identify the particular resource involved in an interaction between components.
- REST relies on the author choosing a resource identifier that best fits the nature of the concept being identified



# RESOURCE REPRESENTATION

- REST components perform actions on a resource by using a **representation** to capture the current or intended state of that resource and transferring that representation between components.
- A representation is a sequence of bytes, plus representation metadata to describe those bytes
  - Document, file, and HTTP message entity, etc
- The data format of a representation is known as a *media type*
- The resource is a conceptual mapping
  - the server receives the identifier (which identifies the mapping) and applies it to its current mapping implementation (usually a combination of collection-specific deep tree traversal and/or hash tables)
  - To find the currently responsible handler implementation
  - The handler implementation then selects the appropriate action+response based on the request content.



# SELF DESCRIPTIVE MESSAGES

- Component interactions occur in the form of dynamically sized messages
- The most frequent form of request semantics is that of retrieving a representation of a resource (e.g., the “GET” method in HTTP)
  - which can often be cached for later reuse
- REST eliminating any need for the server to maintain an awareness of the client state beyond the current request





“**Representational State Transfer**” is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (**state transitions**), resulting in the next page (representing the next state of the application) being **transferred** to the user and rendered for their use..



# NOTES ON REST DEVELOPMENT

- The first edition of REST was developed between October 1994 and August 1995
- REST was the means for communicating Web concepts as during HTTP/1.0 specification and the initial HTTP/1.1 proposal.
- REST was iteratively improved over the next five years and applied to various revisions and extensions of the Web protocol standards.
- REST was originally referred to as the “HTTP object model,” but that name would often lead to misinterpretation of it as the implementation model of an HTTP server.

